On Timely Staging of HPC Job Input Data

Henry M. Monti, *Member*, *IEEE*, Ali R. Butt, *Senior Member*, *IEEE*, and Sudharshan S. Vazhkudai

Abstract—Innovative scientific applications and emerging dense data sources are creating a data deluge for high-end supercomputing systems. Modern applications are often collaborative in nature, with a distributed user base for input and output data sets. Processing such large input data typically involves copying (or *staging*) the data onto the supercomputer's specialized high-speed storage, *scratch* space, for sustained high I/O throughput. This copying is crucial as remotely accessing the data while an application executes results in unnecessary delays and consequently performance degradation. However, the current practice of conservatively staging data as early as possible makes the data vulnerable to storage failures, which may entail restaging and reduced job throughput. To address this, we present a timely staging framework that uses a combination of job start-up time predictions, user-specified volunteer or cloud-based intermediate storage nodes, and decentralized data delivery to coincide input data staging with job start-up. Evaluation of our approach using both PlanetLab and Azure cloud services, as well as simulations based on three years of Jaguar supercomputer (No. 3 in Top500) job logs show as much as 91.0 percent reduction in staging times compared to direct transfers, 75.2 percent reduction in wait time on scratch, and 2.4 percent reduction in usage/hour. (An earlier version of this paper appears in [30].)

Index Terms—High performance data management, data-staging, HPC center serviceability, end-user data delivery

1 Introduction

The advent of extremely powerful computing systems such as Petaflop supercomputers, and the data they can process such as very high resolution space observations, are pushing the envelope on data set sizes. For instance, the large hadron collider [1] or the spallation neutron source [2] will generate petabytes of data. These large data sets are processed by a geographically dispersed user base. Therefore, result output data from high-performance computing (HPC) simulations are not the only source that is driving data set sizes. Input data sizes are also growing many fold [1], [2], [3], [4].

To match the I/O capabilities with the computational power in an HPC center, a job's associated input data is copied or *staged* to a fast local storage at the center—the scratch parallel file system—before the job is started. The use of scratch storage is mandatory, as the alternative of accessing data remotely while a job is executing on (typically) large number of resources creates stalls and wastes precious allotted compute time. Modern applications usually encompass complex analyses, which can involve staging large input data using point-to-point transfer tools such as scp, hsi [5], and GridFTP [6], from observations or experiments. Moreover, the data sources are increasingly becoming dispersed as scientists tackle complex problems, for example, near real-time modeling of adverse weather [7], which depends on distributed sensors. Thus, input data can originate from multiple sources, for example, end-user sites, remote archives (HPSS [8]), Internet repositories (NCBI [9], SDSS [3]), collaborating sites and other centers that run pieces of the job workflow. Therefore, HPC data management is the focus of active research [5], [6], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23] (see Section 1 supplementary material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.279).

Upon submission, a job waits in a *batch queue* at the HPC center to be scheduled, while the input data "waits" on the scratch space. HPC centers are heavily crowded and it is not uncommon for a job to spend hours—or even days on end—in the queue. In the best case when the data is staged at job submission, job completion time, i.e., $(wall\_time + wait\_time)$, is also the time the input data spends on the scratch space. In the worst case, which is more common, the data wait time is longer as users conservatively stage the data much earlier than even job submission. Alternatively, users may also include data movement in the job scripts, which forces the allocated cores to wait for the data to be brought in from remote resources. The time thus wasted is commensurate to $\#cores * time$ to stage in the input data, and is significant in typical allocations.

Scratch space is expensive—costing millions of dollars for state-of-the-art supercomputers, for example, Jaguar's [24] scratch storage has 14,000 disks, 192 object storage servers, 1,300 object storage targets, and 48 controller pairs—and consumes a notable fraction of the HPC center's operations budget. More importantly, scratch space is meant for currently running or soon to run jobs. This usecase precludes simple scratch space management policies, for example, quotas or charging for space usage are rarely used so that currently running jobs will not fail due to lack of space. However, from a center standpoint, suboptimal use of scratch space could impact the center's serviceability, i.e., the ability to serve more incoming jobs. That is why, even with huge scratch space capacities, supercomputer administrators constantly trim usage through purge policies and weekly reminders to users to move their data from scratch storage. From a user standpoint, the input data is exposed to potential unavailability due to storage system failure [25], [26], [27] while it is waiting on the scratch storage.

- *H.M. Monti and A.R. Butt are with the Department of Computer Science, Virginia Tech, 2202 Kraft Drive, Blacksburg, VA 24061. E-mail: {hmonti, butta}cs.vt.edu.*
- *S.S. Vazhkudai is with the Oak Ridge National Laboratory, One Bethel Valley Road, PO Box 2008 MS6016, Oak Ridge, TN 37831. E-mail: vazhkudaiss@ornl.gov.*

Manuscript received 28 Oct. 2011; revised 29 Apr. 2012; accepted 10 Sept. 2012; published online 21 Sept. 2012.
Recommended for acceptance by X.-H. Sun.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-10-0802.
Digital Object Identifier no. 10.1109/TPDS.2012.279.

Consequently, when the job is started, crucial pieces of input data may be unavailable, requiring a rescheduling (delay on the order of hours to days). *What is needed is a framework that enables timely staging of large input data sets for jobs.*

1.1 Design Challenges

Staging the data to be coincident with job startup, i.e., just-in-time (JIT), is challenging. First, we need to know when the user's job will commence. This has been explored extensively [28], [29], and HPC schedulers (e.g., PBS Pro [10], Moab [11]) can also provide a batch queue wait time estimate based on current and historical (jobs with a similar profile) data. However, a simple and direct use of batch queue predictions in JIT staging is not appropriate due to sudden changes in schedules. For example, an unexpected failure can cause a 10,000-core job to suddenly exit, resulting in many jobs being promoted to "ready to run" state, all too quickly.

Second, we need an estimate of how long the data staging would take from the input locations to the HPC center. We need continuous bandwidth measurements so they can be factored in to revise the route dynamically and adapt to changing network conditions. The upshot is that both the queue wait time estimates and network bandwidth estimates are volatile and "soft." Consequently, our staging solution needs to be resilient to adapt to these transient conditions.

1.2 Contributions

This paper makes the following contributions:

Timely data staging framework. We present a JIT staging framework that attempts to have the needed data available at scratch storage just before the job is about to run. To this end, the framework reduces the data copying time to the scratch storage by proactively bringing the data to intermediate storage locations on the path from the end-user site to the HPC center.

Integration of HPC job management and decentralized data transfer systems. We employ an innovative combination of high-efficiency data dissemination (BitTorrent [31]) and network monitoring (NWS [23]) to exploit orthogonal, residual bandwidth and to dynamically adapt to network volatility, respectively, to improve overall scratch space utilization. Further, the overarching unique goal of our work is to reconcile scratch space consumption with volatility (both network and storage) and timely staging, which is in contrast to existing works on decentralized transfers [12], [32], [33], [34], [35].

Use of cloud storage in HPC. We adapt our JIT staging to exploit cloud resources as intermediate storage when available. We demonstrate our solution using Azure [36].

Multipronged evaluation. We evaluate our solutions using both real-world experiments on Azure [36] and PlanetLab [37] as well as extensive simulations using three years worth of job logs from the ORNL Jaguar supercomputer [24].

2 Design

In this section, we describe the goals and the main design of JIT staging, with additional discussion of alternative designs provided in Section 2 of the online supplementary material.

2.1 Objectives

1. *Timely delivery of input data.* Our primary goal is to deliver application input data to center local storage from multiple sources JIT, in the face of both transient network conditions and changing batch queue job wait times. Not properly accounting for such dynamism can have adverse effects on the staging framework: data delivery is delayed, and consequently job turnaround time is increased.
2. *Minimize transfer times.* The ability to minimize transfer times by choosing optimal routes and constantly re-evaluating them is critical for reacting to changes. For instance, this can help handle sudden tightening in the delivery deadline due to an unexpected cancellation of a large job.
3. *Reduce duration of scratch space consumption.* From a center standpoint, it is desirable to stage the data of a waiting job as late as possible so that the scratch space is available for all of the currently running jobs' I/O. Thus, if the waiting jobs' duration of scratch usage is reduced, it would help the HPC center better service the currently running jobs.
4. *Reduce exposure window.* Another downside of staging the data early is its exposure to potential storage system failure. We refer to the time elapsed between when data is staged until the associated job starts running as exposure window, E_w. To protect against storage failures, it is desirable to minimize E_w, preferably as close to 0 as possible. We have shown that minimizing E_w is crucial [15], [30] as disk failure is the norm and not an exception in large supercomputers with tens of thousands of disks. For example, supercomputers such as Jaguar, ASCI Q, ASCI White, and PSC Lemieux all cite storage as a primary reason for system downtime with MTBF of 37.5, 6.5, 40, and 6.5 hours, respectively [38].
5. *Avoid starvation.* Finally, the job scheduler should not become idle because the input data of a waiting job has not been completely staged, as it affects center serviceability.

2.2 Architecture

Fig. 1 shows the high-level system overview, and illustrates interactions between our framework components.

2.2.1 Intermediate Nodes

Our framework uses intermediate nodes (N_is) that can provide temporary storage for data on the path from the source to the HPC center. The intuition behind using N_is is that nodes closer to the center than the user site can support faster data transfers for staging and reduce staging times. This provides for delaying the staging to much later than when using a direct transfer, which also reduces E_w.

2.2.2 Queue Prediction as Staging Deadline

In our design, the HPC center is expected to support a batch queue prediction service (e.g., NWS batch queue prediction [39]), which the users can query before submitting their jobs to get an estimate of queue wait times. Scheduling based on queue wait times is already popular in TeraGrid [17] supercomputer centers. In fact, modern resource managers (e.g., Moab [11]) are beginning to provide services that would enable users to query and obtain start times of queued jobs. The prediction service can usually provide

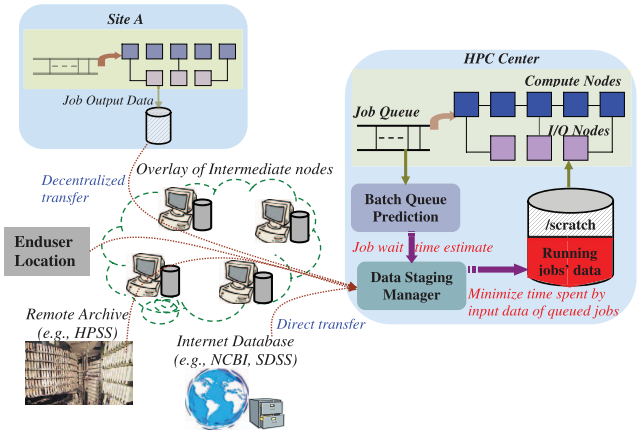


Fig. 1. Overview of the timely staging framework, and interactions between the components.

both wait time estimates as well as the probability of a job starting by a user-specified deadline [39]. In cases where direct wait time predictions are unavailable, the user can pose a query to the service, with a deadline, and determine the likelihood of the job starting by the deadline. A 90 percent or higher probability indicates an affirmation of the user-specified deadline and can be used as the job startup time and, consequently, the staging deadline. However, this deadline is only an estimate as jobs can start earlier due to prediction inaccuracies or other jobs' failures. To accommodate this, we let the *staging manager* allow the user tweak the estimate by up to a fixed factor, f, moving the deadline earlier [30].

2.2.3 Timely Staging Algorithm

Once a deadline for completing the input data staging is determined, the user submits a job script to the staging manager at the center with a description of the job and other details necessary for timely staging. The script includes attributes such as the user-adjusted job startup deadline, the set of intermediate nodes, $<N_i, P_i>$, where P_i denotes the usage properties of the intermediate node N_i, for the decentralized staging process, and the sizes and locations of the input data sets, D_j. The staging manager also takes as input the current snapshot, BW_i, of the observed NWS bandwidth between the HPC center and N_i as well as between the N_is themselves. While we currently use bandwidth snapshots, we note that our model is independent of a specific network "distance" metric and can work with other possible metrics. This could include bandwidth, latency, as well as considerations such as out-of-band agreements.

Algorithm 1 shows the pseudocode for the JIT staging manager. The manager reconciles the predicted job start deadline with the user-adjusted one to determine if it can allow the user's tight deadline. This reconciled deadline is denoted by $T_{JobStartup}$. Based on these parameters, the manager decides upon a data staging schedule, X_j, for each D_j, which delivers the data set in time, $T_j = Min(DirectTransfer, DecentralizedTransfer)$. To estimate these times, the manager uses the measured available bandwidth to the user site as well as the intermediate nodes. To create a distributed schedule, the intermediate nodes are sorted based on available bandwidth and then the number of nodes to which data is sent is increased until overall transfer times that are better than a direct transfer (if possible) can be achieved. This choice is dictated largely by the available bandwidth and storage at the intermediate nodes. When the intermediate nodes can provide a faster transfer, a decentralized transfer is scheduled. Each data set could come from a variety of sources, including those wherein our decentralized transfer software cannot be installed. In such cases, the manager relies on *JIT probes* to the data source to judge if a direct transfer to the HPC center is most appropriate. Alternatively, such input data could also be transferred through the intermediate nodes by having the edge-level nodes pull the data from the source, enabling decentralized staging.

Algorithm 1. The timely staging algorithm.

$Job = CreateJobScript(<N_i, P_i>, D_j, BW_i)$
$T_{Predict} = GetJobStartUpPredictionFromBQP(Job)$
$T_{JobStartup} = ManagerReconcile(Job, T_{Predict}, f)$
for Each D_j **do**
  Determine X_j such that:
  $T_j = Min(DirectTransfer_j, DecentralizedTransfer_j)$
  $ScheduleTransfer(T_j)$
end for
repeat
  $BW'_i = GetNWSUpdate(BW_i)$
  $T'_{JobStartup} = GetBQPUpdate(T_{JobStartup})$
  for Each T_j **do**
    $T'_j = Recalculate(T_j, <N_i, P_i, BW'_i>)$
    if $T'_j > T'_{JobStartup}$ **then**
      Increase the $Fan-in$
    end if
  end for
until Staging Completes

The multi-input staging should obviously also complete before job startup and should satisfy the property, $Max(T_j) \leq T_{JobStartup}$. Minimizing transfer times by choosing the intermediate nodes with the best available transfer rates helps achieve this goal. At the same time, each input staging, X_j, is also started as late as possible to reduce the duration of scratch space consumption and, consequently, the exposure window, E_w of the data sets. The exposure window for each input data set is: $E_{wj} = T_{JobStartup} - T_j$. Then, total exposure of all input data is, $E_w = Sum(E_{wj})$. The closer E_w is to 0, the better. Thus, the ideal start time for each input data set is the one that achieves, $T_{JobStartup} - T_j = 0$. In practice, however, a small difference is desirable to safeguard against unexpected delay. This approach factors in both timely delivery as well as scratch space usage optimization.

2.2.4 Re-Evaluating Staging Decisions

Even after a particular course of action, for example, decentralized transfer, is chosen, the manager periodically re-evaluates the staging (Algorithm 1) based on an updated $<N_i, P_i, BW'_i>$, where BW'_i is the latest snapshot of bandwidth measurements. If the re-evaluated time to staging, T'_j, satisfies the property, $T'_j > T_{JobStartup}$, then, alternate (available) routes are taken to stage the data before job startup, enabling us to meet the staging deadline.

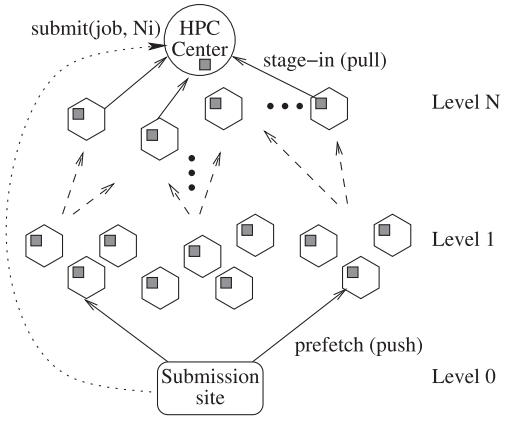


Fig. 2. The data flow path from the client site to the HPC center. Each intermediate node (hexagon) runs NWS (gray square) for bandwidth monitoring.

In addition to re-evaluating the network routes based on updated bandwidth measurements, the staging manager also has to account for batch queue status changes discussed earlier. We address this by having the manager periodically obtain new estimates $T'_{JobStartup}$ from the batch queue service. If the staging schedules reflect that $T_j > T'_{JobStartup}$, then alternate routes are evaluated to ensure timely delivery. A side effect of this is the prevention of job scheduler starvation due to inability to schedule jobs as a result of unfinished stagings.

2.3 Supporting Timely Staging

Once the data staging is initiated, the client chooses a set of nodes from N_is ordered by available bandwidth. Fig. 2 shows the data flow from end-user site to the HPC center. These chosen N_is serve as the Level-1 intermediate nodes. The manager monitors the bandwidths periodically (using NWS) and vary the selected N_is. Next, the input data is split into chunks and parallel transfer of the chunks to Level-1 nodes is initiated. The transfer may also involve further levels of intermediate nodes (up to Level-N). The choice of the number of levels of intermediate nodes is left to the users, and does not have a direct bearing on the center to Level-N node performance that is critical for our design. The levels simply enable users to provide multiple data-flow paths to the center, and we foresee the levels to be not more than two in typical scenarios. Additionally, depending on the availability of intermediate nodes, the client can also stage the data to Level-N nodes much earlier than the deadline.

As the job startup deadline approaches, the proximity of the Level-N nodes to the center allows them to quickly move the input data to the center's scratch space. The JIT manager can vary the fan-in, i.e., the number of Level-N nodes from where to simultaneously retrieve data. The cardinality of the fan-in is chosen to stage all the necessary data before the predicted job start time (Algorithm 1). The fan-in is expanded until the deadline can be met or until no more nodes can be added. The goal is to obtain the best possible transfer time given the intermediate nodes and job deadline. Also, this design allows the Level-N nodes to stage the data at peak (prespecified) bandwidth at the most appropriate time without worrying about the availability (and connection speed) of the submission site (see Fig. 2).

Intermediate nodes provide multiple data-flow paths as well as several alternative options for data delivery. For instance, data may be replicated across different N_is during the transfer from one level to the other. This lets the center to pull data from a number of locations for fault tolerance.

The amount of data transferred between the intermediate nodes will vary depending on the number of nodes used and the above parameters, as well as the network conditions at the time of the transfer. However, as stressed earlier, the intermediate nodes are provided by collaborators (e.g., as in TeraGrid or ESG) that already have an interest in seeing the job succeed, and any overhead due to retransmissions between intermediate nodes can be considered as necessary to this end.

2.4 Discussion

2.4.1 Cloud Resources as Intermediate Nodes

The emerging cloud model can also provide resources for the intermediate sites of our JIT staging. In fact, cloud resources can act as always available geographically distributed locations, *landmark nodes*. A number of cloud features make it suitable for an intermediate staging area. First, the cloud resources are scalable, distributed, and robust, for example, Windows Azure allows blobs (binary large objects) each of up to 50 GB at present [36]. Second, the cloud can provide very high data reliability guarantees through replication, geographically distributed storage, and active fault ramifications. This relieves both HPC centers and end users from expensive data redundancy improving operations. Third, data can be strategically placed in the cloud, i.e., relatively close to an HPC center or end-user, yielding potentially higher transfer rates and lower latency when the data is needed. This is further enhanced if the cloud service provider supports content distribution networks (CDNs). Finally, the cost of utilizing cloud storage resources is very low compared to the multimillion dollar storage systems at HPC centers. This is especially useful for ephemeral collaborations and shorter term projects. In our own work [40], we found that there is a broad range of jobs for which cloud storage and transfer costs are reasonable, especially for a collaboration with several researchers. We further examine the costs of cloud storage Section 3.1.3.

However, using the cloud as an intermediate storage location also has some disadvantages that are addressed by intermediate nodes in this work. First, while the cloud offers many features that can potentially improve performance for data transfers, these are typically static and inflexible (e.g., storing data in the cloud near the HPC center beforehand), or do not match typical HPC workloads (e.g., a CDN requires multiple accesses for improving performance and staging in of job input data does not benefit from this approach). Second, the cloud is a black box and to achieve our goals we must infer through benchmarks and other probes how it will behave under different circumstances, and how to get the best performance for the HPC use-case. For instance, a group of user provided intermediate storage locations are significantly more configurable than a cloud storage location.

TABLE 1
Average Observed Bandwidth between PlanetLab Nodes during Experimentation

	Center	Client	Level-1	Level-2
Center	-	3.82	-	10.9
Client	3.07	-	5.22	-
Level-1	-	3.86	-	4.22
Level-2	9.47	-	5.66	-

All numbers are in Mb/s.

3 Evaluation

In this section, we present an evaluation of our timely data staging using: 1) an implementation (Section 3 of the online supplementary material) running on the PlanetLab testbed [37]; and 2) an HPC center data-subsystem simulator, *simStagein* (Section 4 of the online supplementary material), which is driven by three-year job logs from the Jaguar [24] supercomputer. We also compare our JIT staging to commonly used direct transfer techniques for staging input data in HPC centers. Additional results can be found in Section 5 of the online supplementary material.

3.1 Implementation Results

First, we use the PlanetLab [37] testbed to study the effectiveness of our decentralized staging in a true distributed environment. We chose 20 PlanetLab nodes arranged in a tree-structure: one as the client site and root of the tree, one as the HPC center, 10 and 8 Level-1 and Level-2 nodes (see Fig. 2), respectively. Table 1 shows the average bandwidth observed between the nodes during the course of our experiments. Our results represent averages over a set of three runs.

3.1.1 Decentralized JIT Staging versus Direct Transfer

In this experiment, we compare our decentralized JIT staging to several point-to-point direct transfer tools that are prevalent in HPC: 1) scp, a baseline secure transfer protocol; 2) IBP [19], an advanced transfer protocol that makes storage part of the network, and allows programs to allocate and store data in the network near where they are needed; and 3) GridFTP [41], an extension to the FTP protocol, which provides authentication, parallel transfers, and allows TCP buffer size tuning for high performance; and BBCP [42], which also provides high performance through parallel transfers and TCP buffer tuning. Note that these protocols are all typically supported [43] by HPC centers such as Jaguar [24].

For this experiment, we used a range of file sizes from 1 GB to 5 GB (limited by PlanetLab policies), and measured the time for each direct transfer method between the center and the submission site. For JIT staging, we used a combination of BitTorrent and NWS as outlined earlier.

Table 2 shows the times for the direct data transfer techniques from client to HPC center (scp, IBP, GridFTP, BBCP), from client to Level-1 nodes (Client Offload), and from Level-2 to the center (Center Pull). Compared to a direct transfer, decentralized staging can potentially reduce the last-hop (equivalent to Center Pull) transfer times by 91.0 and 91.0 percent for scp and by 83.9 and 83.4 percent for GridFTP, for 1 GB and 5 GB data sizes, respectively.

TABLE 2
Comparison of Decentralized Transfer Times with Different Direct Transfer Techniques

Step	Transfer time (s)		
	1 GB	2 GB	5 GB
scp	1730	3588	8137
IBP	911	1908	4561
GridFTP	965	1841	4404
BBCP	922	1875	4745
Client Offload	703	1264	4082
Center Pull	155	337	731

The buffer size for IBP, GridFTP, and BBCP is set to 1 MB. The number of streams in GridFTP and BBCP is set to 8 and 16, respectively.

This implies that the decentralized staging can potentially delay copying of data to scratch space by a factor of 11.0 for scp and 5.9 for GridFTP, on average, across the studied file sizes, and still get the data to the center in time for the job to start on time. Thus, JIT staging reduces the time the scratch space has to hold the data, consequently, reducing the exposure window (E_w), and improving center serviceability.

The reported Center Pull time represents the time to transfer the file from Level-1 and Level-2 nodes to the center, and does not include the transfer time from the source. However, the Center Pull is asynchronous, and can start as soon as chunks begin to arrive at Level-2 nodes. We note that the overall transfer time, i.e., the time from when the source starts sending the data to when the center has received all the data is not a suitable metric, as our approach allows the center to delay starting the pull as necessary. However, the earliest time the center can get the input data is still a useful metric. In our system, the center can start retrieving the data as soon as the client has offloaded it to Level-1 nodes. Thus, the Client Offload times reported in Table 2 also serve as the earliest data availability metric, and as stated earlier, are significantly better in our approach compared to a direct data transfer.

3.1.2 Explicit Intermediate Nodes versus Cloud-Based Transfer

For our next experiment, we compare our JIT staging under two scenarios: with explicit non-cloud intermediate nodes, and with a cloud-based data transfer service, CATCH [40], that we have extended from our prior work to be aware of job deadlines. CATCH uses cloud resources, specifically Microsoft Windows Azure [36] cloud storage.

We transferred a 1 GB file using our JIT staging service both with explicit intermediate nodes and CATCH. CATCH utilized 16 streams when transferring data between the cloud and the HPC center. Table 3 shows the result. Both of these numbers assume the data has already been stored either in the cloud or at the intermediate nodes, so only the time to transfer to the center is considered. In these experiments, explicit staging performs significantly better than a Read (Center Pull) operation when using CATCH. However, this does not imply that JIT staging using explicit intermediate nodes is always better. For example, the cloud provides reliability guarantees and performance SLAs. Additionally, this experiment assumes there are a sufficient number of available collaborators to support the explicit intermediate nodes, with reasonable

TABLE 3
Comparison of JIT Staging Using Explicit Intermediate Nodes with CATCH's Cloud-Based Nodes, While Transferring 1 GB

Scheme	Transfer Time (s)
JIT Staging with explicit	155
JIT with CATCH Read (Pull)	548
JIT Staging (50% explicit intermediate nodes)	939

CATCH used 16 streams in this experiment.

TABLE 4
Current Azure Pricing

Storage	$0.125/GB per month based on the daily average usage
CDN	$0.12/GB inbound and outbound
Transfer	$0.00/GB inbound / $0.12/GB outbound

TABLE 5
Cost of Using CATCH for Different Workflows under Varying Pricing Structures

	A	B	C	D
Data size	50 TB	10 TB	1 TB	500 GB
CDN usage	Yes	No	Yes	No
Num. uploads	1	1	1	1
Downloads	10	10	5	10
Cost (current)	$72,250	$13,250	$845	$663
Cost (90%)	$65,025	$11,925	$761	$597
Cost (50%)	$36,125	$6,625	$423	$332
Cost (10%)	$7,225	$1,325	$85	$66

amounts of bandwidth, which may not always be the case. To test this case, we repeat the explicit staging experiment with only half the intermediate nodes, i.e., four nodes. In this case, we observe CATCH performs better than explicit intermediate nodes. CATCH can provide reasonable performance especially when compared to the direct transfer mechanisms and in cases where reliability or the lack of collaborators is an issue. These results suggest that both approaches are viable options for HPC end-user data staging under different scenarios.

3.1.3 Cost of Cloud Usage

In the next experiment, we determine how the cost of cloud services impact CATCH usage. Table 4 shows the current pricing structure of Azure [44]. Table 5 shows three different usage scenarios for HPC application workflows, and the cost of using CATCH for the applications. Note that Azure currently provides free inbound data transfers, which is useful in large scientific projects since it would enable many users to inexpensively upload and modify data collaboratively. This data could then be staged to an HPC center only when needed. Moreover, it is not uncommon for large cloud customers such as an HPC center to receive lower/bulk rates, for example, Netflix using Amazon EC2 [45].

To give a sense of the scale of the job that produces terabytes of data, consider that a 100,000-core run of GTS fusion application on Jaguar produces a 50 TB data set. Since the pricing for cloud usage is expected to fall, the Table also shows the cost of using CATCH if the prices are reduced by 10, 50, and 90 percent. In contrast, consider that in a typical HPC center, the I/O subsystem costs can account for 20 to 30 percent of the acquisition cost and may run into millions of dollars. Even though the acquisition cost is amortized over the life of a machine, the annual running costs can still run into millions of dollars. Information about exact HPC center acquisition and maintenance costs are confidential, and are not made public, thus precluding a direct comparison to CATCH. However, we examine some reasonable estimates (based on our interactions with HPC center managers) in Table 6. We consider a large HPC facility with a 10 PB scratch space and a 5-year life span, and compare three different design choices. We assume 10 percent of the upfront costs as yearly maintenance costs. We note that although the amortized per GB cost can be low, it does not scale with volume. For instance, provisioning a scratch with 50 percent more capacity requires specialized nonstandard hardware to support additional disks, controllers, and so on, not to mention other IT and infrastructure upgrade costs, which would increase the cost per GB by several orders of magnitude, rendering the option ineffective. Moreover, such infrastructure cannot robustly support dynamic workloads, especially for distributed users. While PFS storage is needed at the center for a quick dump of job data, it cannot retain the data beyond a purge window, let alone the duration of a collaboration. Thus, cloud storage can complement storage at the HPC center. What this exercise illustrates is that the HPC center cannot arbitrarily provision the scratch space for dynamic collaborations that may wish to retain TBs of data for the duration of the collaboration. In these cases, it is only fair that such collaborations pay for the cost, rather than having the HPC center pay the price. Moreover, cloud storage can support such collaborative data access more economically than HPC center storage.

TABLE 6
Cost of Provisioning a 10 PB Scratch Space under Differing Cost Structures

	A	B	C
Upfront cost ($)	2 M	4 M	10 M
Yearly cost ($)	200 K	400 K	1 M
Total cost ($)	3 M	6M	15 M
Monthly GB cost ($)	0.005	0.01	0.025

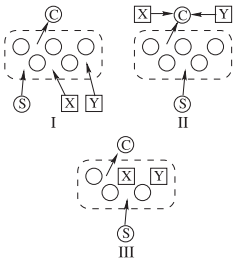


Fig. 3. Configurations used in multi-input test.

3.1.4 Multi-Input Staging

Next, we study the ability of our decentralized staging to accommodate input data from multiple sources. We consider three configurations, as shown in Fig. 3, with two sources (X and Y) of data in addition to the client site (S). In I, the data from all sources is staged in a decentralized manner. This captures retrieving data from slower external sources. In II, we consider fast external sources, for example,

TABLE 7
Comparison of Multi-Input Data Transfer under Direct and Decentralized Staging

Step	Transfer Time (s)		
	Conf I	*Conf II*	*Conf III*
scp	1505	1732	1789
GridFTP	901	946	934
Client Offload (S)	318	672	740
X offload	646	92	N/A
Y Offload	574	142	N/A
Center Pull	312	158	340
Staging time	312	158	340

online data repositories [9] so the center can directly retrieve from them. Finally, in *III*, the intermediate nodes may already have the data, such as collaborating sites in TeraGrid jobs [17]. For each case, we compare scp and GridFTP from the sources to that of our staging. Table 7 shows the results. It is observed that decentralized staging is able to handle multiple sources, and has the potential to outperform the direct transfers by 79.3, 90.8, and 80.9 percent for scp and 65.3, 83.3, 63.6 percent and for GridFTP, in scenario *I*, *II*, and *III*, respectively. In real transfers, the various configurations will switch depending on the transfer rates and staging deadlines.

3.1.5 Behavior under Failures

Improved transfer times are key to JIT staging, and thus reducing scratch space usage times. In the following set of experiments, we study how failures will affect the transfer times under our framework.

First, we examine intermediate node failures. We focus on our decentralized staging, as a failure under direct will result in the data transfer not completing by job startup time, consequently leading to obvious job rescheduling. Fig. 4 shows transfer time achieved by our approach under various failure scenarios, normalized to direct transfer time. We failed two intermediate nodes under three different scenarios: two Level-1 nodes fail, a Level-1 and a Level-2 node fail, and two Level-2 nodes fail. In this test, the number of replicas at each level is set to 3. The system tolerates two Level-1 failures, i.e., 20 percent of Level-1 nodes, with negligible affect. A failure at Level-2 increases the transfer time somewhat (by a factor of 1.3), but two Level-2 failures are significantly more disruptive (time increases by a factor of 2.7). However, this is an extreme case with 25 percent of the Level-2 nodes failing. On the plus side, the transfer time, even with these failures, is less than half (41.2 percent on average) that of the direct transfer. Furthermore, our flexible design can easily accommodate extra replicas to improve fault tolerance, as observed by the reduction of transfer times for each of the Level-2 failure cases when one extra replica is used. This experiment shows that the dynamic rerouting of our approach can adapt to the changing network conditions and ensure meeting the staging deadline with minimal delays, if any. Moreover, the use of a flexible routing path between the client site and HPC center allows for offsetting delays due to intermediate node failures. Moreover, error coding [46] at the source can provide an additional layer of safety, and along with replication through multiple data flow paths can provide good fault tolerance for the staging process.

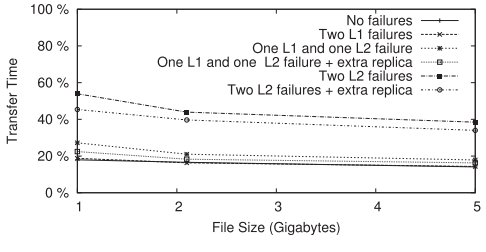


Fig. 4. Normalized transfer time wrt. direct transfer as different combinations of Level 1 (L1) and Level 2 (L2) nodes are failed.

TABLE 8
Statistics About the Job Logs Used in the Simulation Study

Duration	22753 Hrs
Number of jobs	80025
Job execution time	1 s to 120892 s, average 5849 s
Input data size	2.28 MB to 7481 GB, average 65.3 GB

3.2 Log Analysis

In this section, we examine the three-year Jaguar [24] supercomputer job logs in depth to gain information that can improve the implementation of our JIT staging service. Table 8 shows some relevant characteristics of the logs.

3.2.1 Comparing Actual and User-Estimated Job Runtimes

First, we examine the accuracy of user-estimated runtimes, as many works [47] have noted that users generally request more resources than required by their jobs. Fig. 5 plots the user requested runtimes with the actual runtimes as recorded in the logs, and confirms this perception. Across the logs, the users overestimated the requirements by 50.9 times, on average, for jobs longer than 30 seconds (430 times for all jobs), mostly due to jobs ending prematurely. Some of this discrepancy may be due to errors encountered by users while running their jobs, which is pertinent information for our staging service. Nonetheless, much of the difference appears to be users being cautious in specifying requirements, mainly to ensure that their job completes regardless of any transient issues that may occur at the HPC center. As stated earlier, this overestimation works against our overall goals, since we would like to stage user data to the center as late as possible.

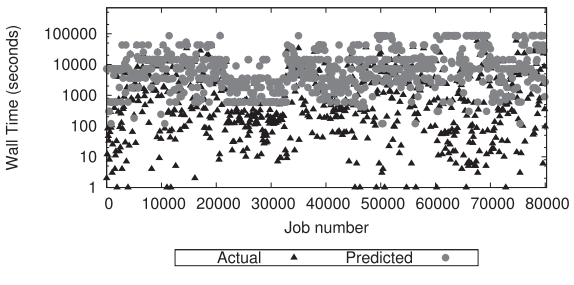


Fig. 5. The actual and user predicted runtimes for each job in the logs. Users typically request significantly more wall time than necessary.

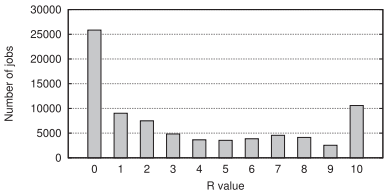


Fig. 6. The R values for each job in the trace grouped into 10 bins. Each bin represents the fraction of the requested wall time actually used.

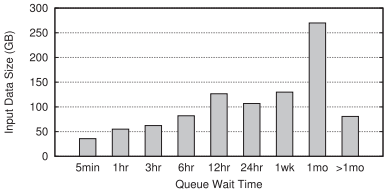


Fig. 7. The effect of queue wait time on average input data size. Each bin value represents the maximum noninclusive wait time for jobs in that bin.

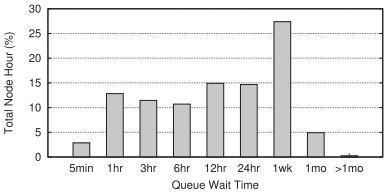


Fig. 8. The effect of queue wait time on utilization (node hour). Each bin value represents the maximum noninclusive wait time for jobs in that bin.

A complementary way of examining actual and user-predicted runtimes is to consider the ratio of actual and user-predicted job runtimes, or R value. The R values can be used to predict accurate job runtimes from user-predicted runtimes [47]. Fig. 6 shows the R values for the studied trace, classified into 10 bins. For example, the 0th bin represents an R value of 0.0 to 0.09, and is equivalent to a job using 0 to 9 percent of its requested time. We observe that most jobs do not run for their requested runtime. For example 32.3 percent jobs use less than 10 percent, and 63.5 percent use less than 50 percent, while only 16.4 percent use more than 90 percent of the requested allotment. There are also a small (but significant) number of jobs that use more than 100 percent of their predicted time. We presume that these are either completed jobs that spend a few extra seconds freeing resources, or jobs that have encountered errors. Out of 10,584 (13.2 percent) jobs that use more than their allocation, 890 run 2 minutes past the requested allocation time and only 141 run 5 minutes past the requested allocation time.

3.2.2 Examining Trends in Queue Wait Times

In Section 2, we discussed using a batch queue prediction service to provide staging deadlines, however, the estimates provided by these services are primarily estimates of jobs' queue wait times. Queue wait times are particularly important to our JIT staging as a job must have sufficient queue wait time left for the required data to be staged in on time. Moreover, understanding the relationship between queue wait time and other important center metrics could provide further insights for refining the design parameters of JIT staging.

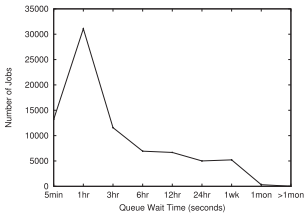


Fig. 9. The number of jobs in each bin.

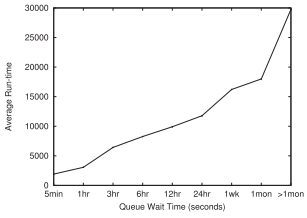


Fig. 10. The effect of queue wait time on average job runtime.

First, we examine how a job's input data size is associated with queue wait times. Fig. 7 shows the result for queue wait times ranging from under 5 minutes to over 1 month. It is observed that the average input data size grows slower than the queue wait time. For example, a job that spends 3 hours waiting in the queue has an average input data size of 62.2 GB, while a job that spends 6 hours (100 percent increase) in the queue has an average of 31.8 percent more input data or 82.0 GB. This trend is similar at longer queue wait times as well.

However, a fundamental assumption in our analysis and simulations is that the *input data sizes = num cores * memory used*. Larger jobs frequently mean more cores and memory, but they may not always mean more data. Jobs can simply be large simulations that use very little data but produce large outputs. On the contrary, data analysis jobs are usually not that large in terms of cores and memory, but consume large amounts of input data. Unfortunately, the information in the job logs does not allow making this distinction. Nevertheless, the logs suggest that for medium to large sized jobs, enough time is spent waiting in the queue before running to provide an opportunity to do JIT staging of job data.

Second, we examine the association of queue wait times and center utilization expressed as a fraction of total node hours as shown in Fig. 8. Here, the *node hours* are defined as $walltime * number\ of\ nodes$. For these logs, jobs that spend long periods of time waiting in the queue use significant center resources. The single bin with the largest amount of node hours used is <1 week (longer than 1 day) with 17.8 m node hours or 27.4 percent of the total center utilization. Another observation is that jobs that spend more than 12 hours in the queue account for 62.1 percent of the overall utilization of the center, even though they only account for 21.5 percent of all jobs. This analysis suggests that jobs that use significant resources are also the ones most able to take advantage of our JIT staging service.

Finally, a range of important trends relating to queue wait time, such as runtime, and average utilization are plotted in Figs. 9, 10, 11, and 12. We note that these trends may not be applicable for other logs and HPC installations, but provide useful insights in realizing a JIT data staging service.

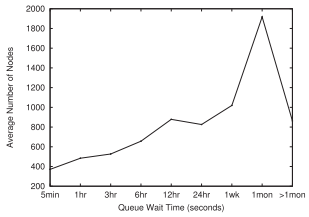


Fig. 11. The effect of queue wait time on the average number of nodes used for a job.

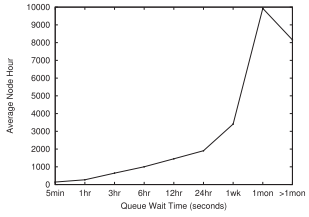


Fig. 12. The effect of queue wait time on average utilization (node hour).

TABLE 9
The Number of Jobs Impacted by Unexpected Job Failures

Time Window	Jobs Affected				
	Total	Avg.	Max.	Min.	Absolute Total
30 s	10049	1.52	20	0	9422
1 min	17686	1.89	40	0	14869
3 min	37988	3.18	210	0	24397
5 min	53964	4.09	210	0	29551

3.2.3 Quantifying the Effect of Unexpected Job Failures

Unexpected job failure may cause jobs waiting in the queue to start executing immediately, (much) earlier than their planned/predicted start times. Such unexpected job execution poses potential problems for JIT staging, as the staging of a job's associated data may have not yet completed. In this experiment, we analyze the job logs to quantify the impact of unexpected job failures by measuring both how frequently they occur and how much time the waiting jobs spend in the queue prior to running.

We step through the logs and examine each job individually. If a job does not use its entire requested allocation, we treat it as a potential failure, and observe all of the subsequent newly running jobs that start in a short time window after the initial failure. The time windows examined range from 30 seconds to 5 minutes. This approach assumes that every job starting in the time window was directly affected by the failing job and is likely to overestimate the number of new jobs. We count the total number of jobs affected by failures and the average, maximum, and minimum number of new jobs per failure. The results of this analysis are shown in Table 9. Since each job is examined individually, there is the potential to count the same new job multiple times. To quantify the impact of such duplication, we also included the "Absolute Total," which removes any duplicate jobs that have been counted several times. On average, each job failure causes only a few new jobs to start unexpectedly, from 1.52 to 4.09 jobs. However, for the larger time windows, there can be up 210 new jobs that appear to be affected. Overall, anywhere from 11.8 to 36.9 percent of all jobs can be affected by job failures, depending on the time window.

TABLE 10
Input Data Size and Amount of Queue Wait Time of Jobs Affected by the Failure of Other Jobs

Time Window	Queue Wait (Min)			Data Size (GB)		
	Avg.	Max.	Min.	Avg.	Max.	Min.
30 s	303.4	72726	1.67	66.4	7481.7	0.24
1 min	341.2	72726	1.67	67.4	7481.7	0.24
3 min	326.4	72726	1.67	70.4	7481.7	0.24
5 min	312.6	72726	1.67	69.2	7481.7	0.24

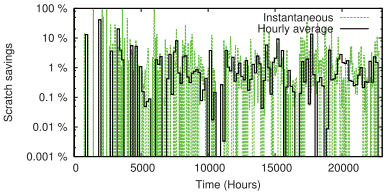


Fig. 13. Scratch savings under timely staging compared to direct. Purge period is seven days.

We also examined the amount of time jobs spend in the queue and the input data sizes to see how much opportunity there is for our JIT staging service. The results can be seen in Table 10. The analysis converges to the top and bottom values quickly for both queue wait time and input data sizes. The average queue wait times range from 303.4 to 341.2 minutes, while the average data sizes range from 66.4 to 70.4 GB. It seems unlikely that these are general results, but they emphasize that a JIT time staging service must be able to handle job failures.

3.3 Simulation Results

In this section, we use the job logs discussed above to study the performance of timely staging using *simStagein*.

3.3.1 Impact on Scratch Space Usage

In this experiment, we quantify the impact of timely staging on scratch space usage. We play the logs in our simulator and determine the amount of scratch used both under direct and timely staging. For this test, we assume that the scratch is empty at the beginning, and use perfect batch queue prediction. Moreover, the center is setup for weekly purges of the scratch space and the maximum center in-bound bandwidth is limited to 10 Gb/s. Only input data is considered, and a data item is only purged if its associated job has completed. Fig. 13 shows the instantaneous savings in scratch space usage by timely staging compared to direct, measured every 10 minutes. The instantaneous savings (associated with a job input data) become zero as the job startup time approaches, as timely staging has to bring in the necessary data. A more representative aspect is the average savings over a period of time, as it captures not only the savings but the duration for which the savings were possible. Therefore, we also show the average savings calculated per hour. Finally, we calculated the average savings per hour across the entire log, and found that staging potentially uses 2.43 percent less scratch per unit of time (e.g., 24.9 GB/hour on average per Terabyte of storage) compared to direct. Thus, timely staging is a promising way for conserving precious scratch resource.

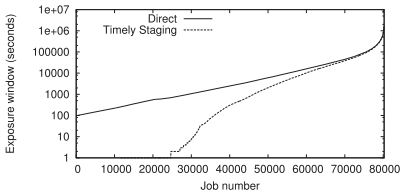


Fig. 14. Size of exposure window for each job in the log.

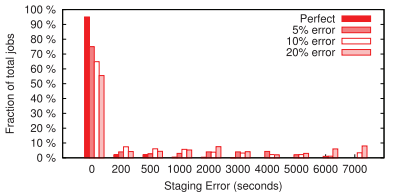


Fig. 15. The distribution of staging error under different batch queue prediction accuracy.

3.3.2 Effect on Exposure Window

In this experiment, we repeat the previous experiment, but now study the exposure window (E_w), i.e., duration for which the data has to wait on the scratch before the associated job is run. Fig. 14 shows the observed E_w under direct and timely staging, for each job in our log, arranged in ascending order. In this experiment, timely staging can potentially reduce the E_w of 30.7 percent of the jobs to zero, and for the remaining jobs it was capable of reducing E_w by 64.2 percent, i.e., 75.2 percent reduction on average across all jobs. Moreover, we found that E_w was reduced by more than a factor of 10 for 48.3 percent of the jobs. However, it is seen that some jobs ($\approx$1.3%) with large E_ws saw only negligible (<1%) affect from timely staging. The reason for this is that: 1) many jobs require large input data, so the long duration of transfer increases the effective E_w; and 2) many jobs in our logs arrived in bursts, and timely staging is forced to start transfers early to ensure all necessary data is available and avoid staging errors. Overall, the significantly reduced E_w for most jobs under JIT staging shows that it can provide better resiliency against storage system failures and costly restaging.

3.3.3 Effect of Job Startup Time Prediction

In this experiment, we randomly introduce up to 20 percent variance in the batch queue prediction and the actual job start-up time. Then, we simulate the time by which timely staging will miss the actual job start-up, i.e., staging error. Fig. 15 shows the distribution of staging error for different prediction accuracies. The results show the dependence of timely staging on the accuracy of batch queue prediction: as the error in accuracy increases from 0 to 20 percent, the number of jobs with no staging error reduces from 95 to 75 percent, i.e., by 21 percent. However, even with increased prediction error, the number of jobs with significant delays is much less than half (30.6 percent of the jobs suffer a staging error of more than 1000 seconds). Note that in this test, we assumed that the prediction error remains constant, however, in real scenarios, the accuracy is improved as the start-up time draws near, implying that timely staging will have much improved performance than studied in this case. Finally, the results show that the approach can withstand some prediction errors, and with improved predictions becoming available, can provide better staging alternatives.

4 Conclusion

In this paper, we have presented the design and implementation of a timely staging framework to coincide input data delivery with job startup. Our framework leverages periodic job wait time estimates from a batch queue prediction service, user-specified intermediate nodes, cloud storage, and periodic network bandwidth measurements to deliver input data on time. Our evaluation shows as much as 91.0 percent reduction in staging times compared to direct transfers, 75.2 percent reduction in wait time on scratch, and 2.4 percent reduction in usage/hour. Thus, our JIT staging solution is able to reconcile several key factors to reduce the duration of scratch space consumption and the exposure window, and adapt to volatility to deliver the data on time, consequently improving HPC center serviceability.

Acknowledgments

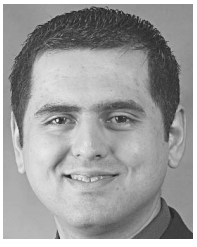
This research was sponsored by the LDRD Program, and NCCS of ORNL, managed by UT-Battelle, LLC for the US DOE under contract no. DE-AC05-00OR22725, and by the US NSF Awards CCF-0746832, CNS-1016793, and CNS-1016408.

References

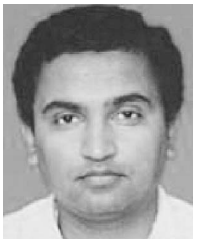
[1] "Large Hadron Collider," http://lhc.web.cern.ch/lhc/, 2013.
[2] "Spallation Neutron Source," http://www.sns.gov/, 2008.
[3] "Sloan Digital Sky Survey," www.sdss.org, 2005.
[4] "Laser Intrfirom, Gravt.-Wave Obsrv," http://www.ligo.caltech.edu/, 2008.
[5] M. Gleicher, "HSI: Hierarchical Storage Interface for HPSS," http://www.hpss-collaboration.org/hpss/HSI/, 2011.
[6] W. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link, "The Globus Striped GridFTP Framework and Server," *Proc. ACM/IEEE Conf. Supercomputing (Supercomputing)*, 2001.
[7] *Near Real Time Modeling of Weather, Air Pollution, and Health Outcome Indicators in New York City*, http://cfpub.epa.gov/ncer_abstracts/index.cfm/fuseaction/display.abstra ctDetail/abstract/8649, 2012.
[8] R.A. Coyne and R.W. Watson, "The Parallel I/O Architecture of the High-Performance Storage System (HPSS)," *Proc. IEEE 14th Symp. Mass Storage Systems (MSS)*, 1995.
[9] *Nat'l Center for Biotech Info*, http://www.ncbi.nlm.nih.gov/, 2013.
[10] *Pbs Pro Technical Overview: Scheduling and File Staging*, https://secure.altair.com/sched_staging.html, 2008.
[11] *Cluster Resources Inc.*, http://www.clusterresources.com/, 2008.
[12] T. Kosar and M. Livny, "Stork: Making Data Placement a First Class Citizen in the Grid," *Proc. 24th Int'l Conf. Distributed Computing Systems (ICDCS)*, 2004.
[13] J. Bent, D. Thain, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny, "Explicit Control in a BADFS," *Proc. First Conf. Networked Systems Design and Implementation (NSDI)*, 2004.
[14] Z. Zhang, C. Wang, S.S. Vazhkudai, X. Ma, G. Pike, J. Cobb, and F. Mueller, "Optimizing Center Performance through Coordinated Data Staging Scheduling and Recovery," *Proc. ACM/IEEE Conf. Supercomputing (SC)*, 2007.
[15] H. Monti, A.R. Butt, and S.S. Vazhkudai, "Scratch as a Cache: Rethinking HPC Center Scratch Storage," *Proc. 23rd Int'l Conf. Supercomputing (ICS)*, 2009.
[16] "DMOVER: Scheduled Data Transfer for Distributed Computational Workflows," http://www.psc.edu/general/software/packages/dmover/, 2008.

[17] "TeraGrid," *Grid Infrastructure Group*, http://www.teragrid.org/, 2013.
[18] D. Thain, S.S.J. Basney, and M. Livny, "The Kangaroo Approach to Data Movement on the Grid," *Proc. IEEE 10th Int'l Symp. High Performance Distributed Computing (HPDC)*, 2001.
[19] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski, "The Internet Backplane Protocol: Storage in the Network," *Proc. Network Storage Symp.*, 1999.
[20] V. Bhat, S. Klasky, S. Atchley, M. Beck, D. Mccune, and M. Parashar, "High Performance Threaded Data Streaming for Large Scale Simulations," *Proc. IEEE/ACM Fifth Int'l Workshop Grid Computing (Grid)*, 2004.
[21] P. Rizk, C. Kiddle, and R. Simmonds, "A Gridftp Overlay Network Service," *Proc. IEEE/ACM Seventh Int'l Conf. Grid Computing(Grid)*, 2007.
[22] G. Khanna, U. Catalyurek, T. Kurc, R. Kettimuthu, P. Sadayappan, I. Foster, and J. Saltz, "Using Overlays for Efficient Data Transfer over Shared Wide-Area Networks," *Proc. ACM/IEEE Conf. Supercomputing*, 2008.
[23] R. Wolski, N. Spring, and J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," *Future Generation Computing Systems*, vol. 15, no. 5 pp. 757-768, 1999.
[24] *Nat'l Center for Computational Sciences*, http://www.nccs.gov/, 2013.
[25] B. Schroeder and G. Gibson, "Disk Failures in the Real World: What Does an MTTF of 1000,000 Hours Mean to you?," *Proc. Fifth USENIX Conf. File and Storage Technologies (FAST)*, 2007.
[26] E. Pinheiro, W.-D. Weber, and L. André Barroso, "Failure Trends in a Large Disk Drive Population," *Proc. Fifth USENIX Conf. File and Storage Technologies (FAST)*, 2007.
[27] S. Shah and J.G. Elerath, "Reliability Analysis of Disk Drive Failure Mechanisms," *Proc. Symp. Ann. Reliability and Maintainability (RAMS)*, 2005.
[28] W. Smith, V. Taylor, and I. Foster, "Using Runtime Predictions to Estimate Queue Wait Times and Improve Scheduler PERF," *Proc. Job Scheduling Strategies for Parallel Processing (JSSPP)*, 1997.
[29] A. Downey, "Using Queue Time Predictions for Processor Allocation," *Proc. Job Scheduling Strategies for Parallel Processing (JSSPP)*, 1997.
[30] H. Monti, A.R. Butt, and S.S. Vazhkudai, "Reconciling Scratch Space Consumption Exposure, and Volatility to Achieve Timely Staging of Job Input Data," *Proc. IEEE Int'l Symp. Parallel and Distributed Processing (IPDPS)*, 2010.
[31] B. Cohen, "BitTorrent Protocol Specification," http://www.bittorrent.org/protocol.html. May 2007.
[32] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High b/w Data Dissemination Using an Overlay Mesh," *Proc. 19th ACM Symp. Operating Systems Principles (SOSP)*, 2003.
[33] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat, "Using Random Subsets to Build Scalable Net. Services," *Proc. Fourth Conf. USENIX Symp. Internet Technologies and Systems (USITS)*, 2003.
[34] S. Annapureddy, M.J. Freedman, and D. Mazires, "Shark: Scaling File Servers via Cooperative Caching," *Proc. Second Conf. Networked Systems Design and Implementation (NSDI)*, 2005.
[35] K. Park and V.S. Pai, "Scale and Performance in the CoBlitz Large-File Distribution Service," *Proc. Third Conf. Networked Systems Design and Implementation (NSDI)*, 2006.
[36] *MS Azure*, http://www.microsoft.com/windowsazure/, 2010.
[37] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A Blueprint for Introducing Disruptive Technology into the Internet," *Proc. First Workshop Hot Topics in Networks (HotNets)*, 2002.
[38] C. Hsu and W. Feng, "A Power-Aware Run-Time System for High-Performance Computing," *Proc. ACM/IEEE Conf. Supercomputing (SC)*, 2005.
[39] *Batch Queue Prediction*, http://nws.cs.ucsb.edu/ewiki/nws.php?id=Batch+Queue+Prediction, Sept. 2008.
[40] H. Monti, A.R. Butt, and S.S. Vazhkudai, "CATCH: A Cloud-Based Adaptive Data Transfer Service for HPC," *Proc. IEEE Int'l Symp. Parallel and Distributed Processing (IPDPS)*, 2011.
[41] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke, "GASS: A Data Movement and Access Service for Wide Area Computing Systems," *Proc. Workshop I/O in Parallel and Distributed Systems*, 1999.
[42] *Bbcp*, http://www.slac.stanford.edu/, 2010.
[43] *Nccs User Support - Data Transfer*, http://www.nccs.gov/user-support/general-support/data-transfer/, 2010.
[44] "Windows Azure Pricing," *Microsoft*, http://www.microsoft.com/windowsazure/pricing/, June 2010.
[45] *The Netflix Tech Blog*, http://techblog.netflix.com/2010/12/four-reasons-we-choose-amazons-clou d-as.html, 2012.
[46] J. Plank, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-Like Systems," *Software - Practice and Experience*, vol. 27, no. 9 pp. 995-1012, 1997.
[47] W. Tang, N. Desai, D. Buettner, and Z. Lan, "Analyzing and Adjusting User Runtime Estimates to Improve Job Scheduling on the Blue Gene/P," *Proc. IEEE Int'l Symp. Parallel and Distributed Processing (IPDPS)*, 2010.

Henry M. Monti received the BS degree in computer science from George Mason University, Virginia, in 2006. He is currently working toward the PhD degree in computer science and applications at Virginia Polytechnic Institute and State University. His research interests include high-performance computing, distributed systems, and cloud computing. He is a member of IEEE.



Ali R. Butt received the BSc (Hons) degree in electrical engineering from the University of Engineering and Technology Lahore, Pakistan, in 2000 and the PhD degree in electrical and computer engineering from Purdue University in 2006. At Purdue, he also served as the president of the Electrical and Computer Engineering Graduate Student Association for 2003 and 2004. He is an associate professor of computer science at Virginia Tech. His research interests include experimental computer systems, especially in data-intensive high-performance computing (HPC) and the impact of technologies such as massive multicores, cloud computing, and asymmetric architectures on HPC. His current work focuses on I/O and storage issues faced in modern HPC systems. He is a recipient of the NSF CAREER Award (2008), an IBM Faculty Award (2008), an IBM Shared University Research Award (2009), a Virginia Tech College of Engineering "Outstanding New Assistant Professor" Award (2009), a best paper award (MASCOTS 2009), and a NetApp Faculty Fellowship (2011). He was an invited participant (2009) and an organizer (2010) for the NAE's US Frontiers of Engineering Symposium. He is a member of USENIX and ASEE, and a senior member of the ACM and IEEE.



Sudharshan S. Vazhkudai received the doctorate degree from The University of Mississippi in 2003 and performed his research at Argonne National Laboratory. He is a research scientist in the Computer Science and Mathematics Division at Oak Ridge National Laboratory, a US Department of Energy facility. In addition, he is also a joint faculty associate professor at The University of Tennessee. He is broadly interested in storage systems, HPC I/O architectures and distributed computing.